



Hex-dominant meshing: mind the gap!

Nicolas Ray, Dmitry Sokolov, Maxence Reberol, Franck Ledoux, Bruno Lévy

► To cite this version:

Nicolas Ray, Dmitry Sokolov, Maxence Reberol, Franck Ledoux, Bruno Lévy. Hex-dominant meshing: mind the gap!. SPM 2018 - International Conference on Solid and Physical Modeling, Jun 2018, Bilbao, Spain. hal-01927557

HAL Id: hal-01927557

<https://inria.hal.science/hal-01927557>

Submitted on 19 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hex-dominant meshing: mind the gap!

Nicolas Ray^a, Dmitry Sokolov^a, Maxence Reberol^a, Franck Ledoux^b, Bruno Lévy^a

^aUniversité de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

^bCEA, DAM, DIF, F-91297 Arpajon, France

Abstract

We propose a robust pipeline that can generate hex-dominant meshes from any global parameterization of a tetrahedral mesh. We focus on robustness in order to be able to benchmark different parameterizations on a large database. Our main contribution is a new method that integrates the hexahedra (extracted from the parameterization) into the original object. The main difficulty is to produce the boundary of the result, composed of both faces of hexahedra and tetrahedra. Obviously, this surface must be a good approximation of the original object but, more importantly, it must be possible to remesh the volume bounded by this surface minus the extracted hexahedra (called void). We enforce these properties by carefully tracking and eliminating all possibilities of failure at each step of our pipeline.

We tested our method on a large collection of objects (200+) with different settings. In most cases, we obtained results of very good quality as compared to the state-of-the-art solutions. To ease reproducing our results and benchmarks, we provide a C++ implementation of the pipeline in the supplemental materials.

Keywords: hex meshing, parameterization, mesh extraction

Introduction and previous work

We introduce a pipeline that takes as input a tetrahedral mesh and that produces a hexahedral dominant mesh. Hexahedral-dominant meshing is a relaxation of full hexahedral meshing, that allows to greatly improve robustness at the expense of introducing a small number of tetrahedra. We focus here on the robustness of hex-dominant meshing, and on its robustness, where most difficulties come from mixing and matches meshes with incompatible discretizations.

Classic hexahedral meshing. Hexahedral meshing generates meshes composed of deformed cubes (hexahedra). Such meshes are often used for simulating some physics (deformation mechanics, fluid dynamics ...) because they can significantly improve both speed and accuracy. This is because (1) they contain a smaller number of elements (5-6 tetrahedra for a single hexahedron), (2) the associated tri-linear function basis has cubic terms that can better capture higher order variations, (3) they avoid the locking phenomena encountered with tetrahedra [1], and (4) hexahedral layers can be aligned along geometric boundary features and/or some physical characteristics (flow direction, shock wave, heat gradient...). Fulfilling those criteria required to generate hexahedral block-structured meshes. Despite 30 years of research efforts and important advances, mainly led by Sandia National Labs in the U.S. [22, 23], generating block-structured hexahedral meshes still requires considerable manual intervention in most cases (days, often weeks for the most complicated domains). Some methods [14, 24] constrain the boundary

into a regular grid, but they are not fully satisfactory either, since the grid is not aligned with the boundary. In the past two decades, numerous approaches for full hexahedral meshing were proposed, geometric as well as topological. Geometric approaches comprise methods such as plastering [2] and H-Morph [18], whereas topological approaches include whisker weaving [22], recursive bisection [3] and dual cycle elimination [16]. Unfortunately, it is very easy to find failure cases for each of these methods: all of them can fail in the termination process. For geometric approaches, it typically happens when advancing fronts meet at the medial axis.

Hexahedral meshing by global parameterization.

To overcome this problem, several approaches based on global parameterizations were recently proposed [7, 17]. Since they use *global* optimizations instead of advancing fronts with *local* decisions, these methods do not introduce any discontinuity on the medial axis. Global optimization produces impressive results, but it has failure cases of its own. For the sake of completeness, we also mention the alternative approach proposed in [8], that generates a hexahedral mesh by merging tetrahedra of an input mesh along the guidance direction field. All these methods (parameterizations and agglomeration) rely on frame fields [10, 19], however, no existing frame-field method is guaranteed to produce an *integrable* 3D frame field. There were some attempts to preprocess frame fields [12, 11], but unfortunately they do not address all possible issues. Locally modifying the input frame field reduces the number of failure cases, but many of them remain unsolved. Another

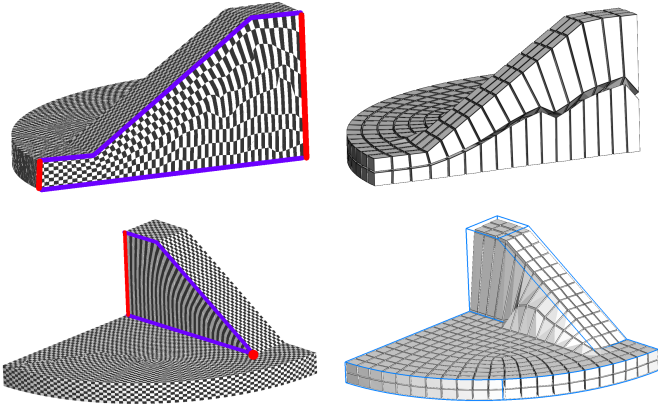


Figure 1: Left: in the presence of triangular wedges, relations between constraints may create a highly distorted parameterization. Right: “HexEx” successfully constructs a valid hexahedral mesh even with such a degenerate parameterization, but clearly, the distorted elements still need to be fixed. Note also how the input geometry was deformed (lower right).

possible option is to relax some unnecessary expectations that concern the parameterization while recovering hexahedra, as done in [13]. In most cases this recovers a correct mesh around singularities.

However, as mentioned in the discussion section of [13], some configurations cannot be handled by HexEx. In Figure 1, we show a “triangular wedge”, often present in mechanical parts. Due to the network of constraints (in blue), the two red edges are supposed to have the same length in parameter space, leading to highly stretched elements. On the bottom row, the edge in red is constrained to have the same length as the red point (i.e. zero), thus generating a single row of extremely stretched elements. Thus, it is in general not possible to extract a hex-dominant mesh of good quality from the sole parameterization. The parameterization needs to be complemented with an algorithm that isolates the distorted zones and remeshes them. Note also that HexEx deformed the geometry around the degeneracy (lower-right). To avoid this behavior, a way of controlling the Hausdorff distance is also needed.

Hexahedral dominant remeshing. We think that given the current state of the art, aiming at full-hexahedral meshing in the general case is not realistic yet¹. For this reason, we focus on hexahedral-dominant meshing instead of full-hex, with the aim of bringing the proportion of hexahedra as near to 100% as possible.

Starting from a tetrahedral mesh, we first compute a global parameterization. Integer iso-values of the global parameterization define a deformed grid inside the volume (Figure 2 a). It is in general not possible to extract from the parameterization a grid that fills the entire volume. Therefore, we extract hexahedra from the regular part of the parameterization, as well as the boundary of the remaining volume which have a singular/degenerate param-

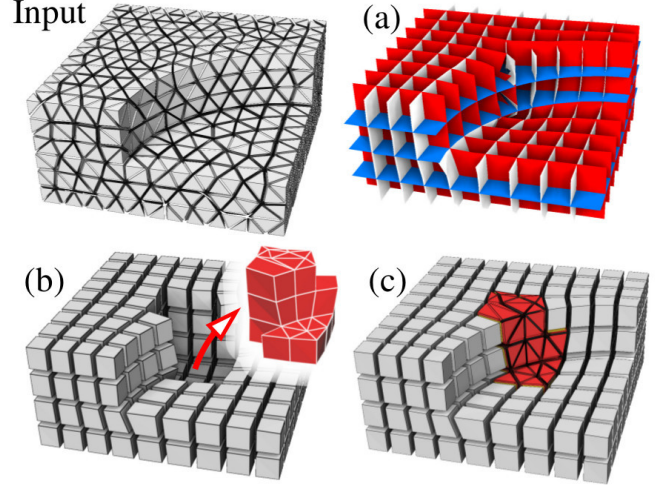


Figure 2: Our hexahedral-dominant meshing procedure: Start from an input tetrahedral mesh. Compute a global parameterization (a). Extract hexahedra by contouring the isovalues of the parameterization. Isolate the boundary of the void (in red), i.e., the volume with a degenerate / singular parameterization (b), shown in red. Remesh the void and stitch it into the hexahedral mesh (c).

eterization (Figure 2 b). Finally, we remesh the remaining void and stitch the meshes together (Figure 2 c).

Our main technical contribution is to integrate hexahedral faces (quads) in the original object boundary (triangular mesh) without making it impossible to remesh the void. The idea to fill the remaining void of hexahedral mesh by tetrahedra was used in front propagation methods about 20 years ago [15]. However, it was in much simpler settings, where the mesh of the input’s boundary was directly the final quad mesh. Note that even in this case the algorithm was not robust, it failed in more than 10% runs.

We aim at bringing both the hex proportion and the robustness as close to 100% as possible. For this purpose, we combine the global structure obtained by global parameterization approaches (e.g. CubeCover [17] and PGP3D [21]) with the robustness of tetrahedral remeshing algorithms (TetGen [20]). Our results are identical to the state of the art in the zones where the global parameterization successfully captures the model to be remeshed. It just fills other zones by tets instead of nothing.

Our contributions. Our contribution is two-fold: we propose a robust *technical solution* that identifies and remeshes parts of models where the result of the global parameterization method is not satisfying. We also provide a complete *testing environment* that allows to compare existing frame field generation and global parameterization algorithms.

Technical solution. Global parameterization generates hexahedra in a large part of the input volume, that is where the parameterization is not degenerate. The remaining void conceptually corresponds to the (volumetric)

¹with the exception of the octree-based approach in [14] that is a robust/valid/efficient solution if boundary alignment is not required.

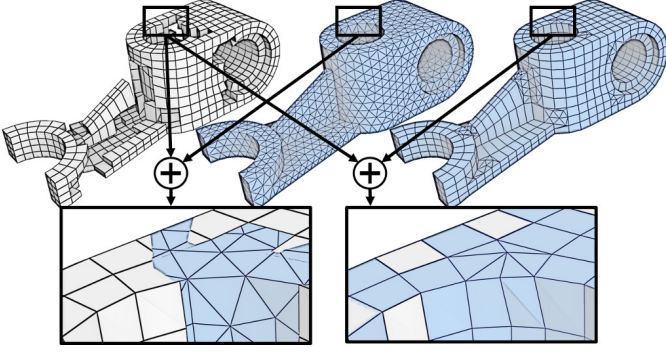


Figure 3: Left: boundary of the hexahedral mesh. Middle: incompatibility with the input boundary (blue). Right: compatibility with our quad dominant remeshing (blue).

boolean difference between the input volume and the hexahedra generated from the parameterization. However, it can not be directly computed this way because the boundaries do not match (see Figure 3).

We instead compute this void by working on its boundary. It defines the void boundary as the difference between the object boundary and the hexahedra mesh boundary. To be able to place hexahedra on the object boundary, we first remesh the object boundary by introducing quads that are likely to match hexahedral facets. The robustness difficulty is to guarantee that the void can always be computed from the surface we manipulate. Similar problems (with a union operation) are still open for mesh assembly [6, 5] but, in our context, we can exploit the mapping between quads and triangles.

Our solution is able to control the Hausdorff distance to the original mesh, and is very robust, as discussed in §7.

Testing environment. We complement this algorithm with a processing pipeline that decomposes hexahedral-dominant meshing into independent steps. Existing remeshing algorithms as well as the corresponding software involve a long sequence of steps orchestrated within a complicated architecture, and no existing research article of reasonable length can address all of them in a self-contained and complete manner. It makes it difficult to reproduce previous results, and to evaluate the impact of improving one step of the pipeline. For instance, if one wants to work on the quality of frame field generation (e.g. [11]), it is not possible to focus on the extraction of hexahedra from the parameterization (e.g. [13]) in the same research article.

Our open-source implementation in the supplemental material will help comparing existing and future algorithms for each step of the pipeline (3D direction field generation, global parameterization, hexahedral mesh extraction). More importantly, it will make it possible to focus on each part of the pipeline *independently*, and to evaluate its impact on the quality of the output.

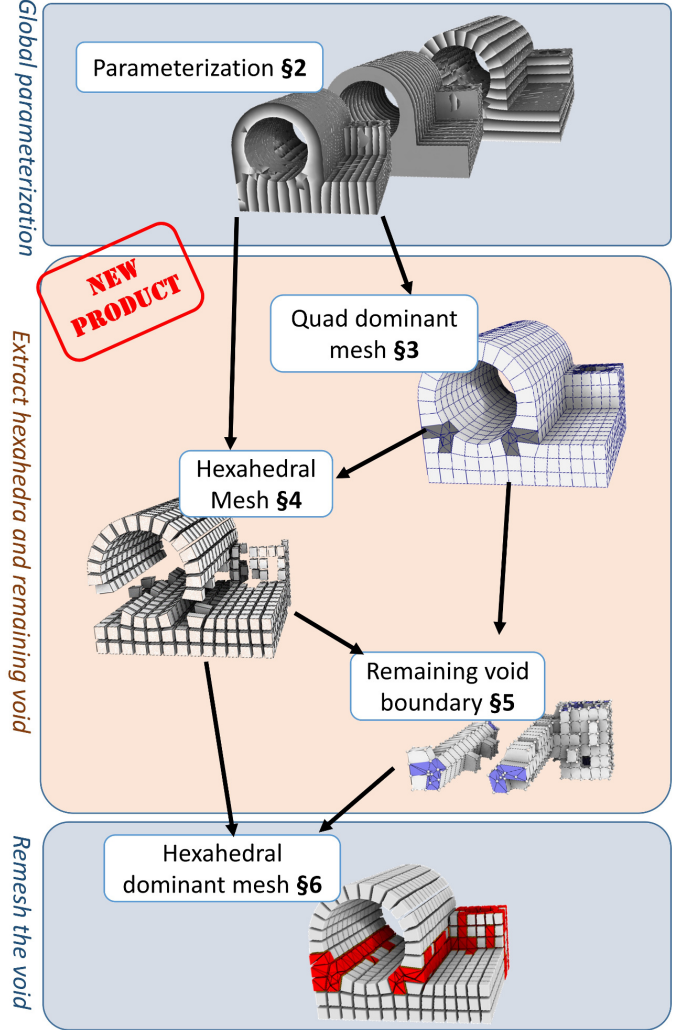


Figure 4: Overview of the pipeline with the result of each step.

1. Overview of the hex-dominant meshing pipeline

Our hexahedral remeshing pipeline is decomposed into three main steps (refer to Figures 2 and 4):

- compute a global parameterization (§2),
- generate hexahedra and extract the remaining void:
 - §3: we compute a quad-dominant mesh from the input boundary. The goal is to make it compatible with the hexahedral mesh that we compute in the subsequent step; we generate quads that are likely to match hexahedra faces.
 - §4: we extract the hexahedra from the pre-image of the parameterization (i.e., the deformed grid shown in Figure 2 a). From this set of hexahedra, we remove the elements that self-intersects or intersects the quad-dominant mesh of the boundary;
 - §5: we compute the boolean difference between the surface of the hexahedral mesh and the quad-dominant mesh of the boundary. Since the

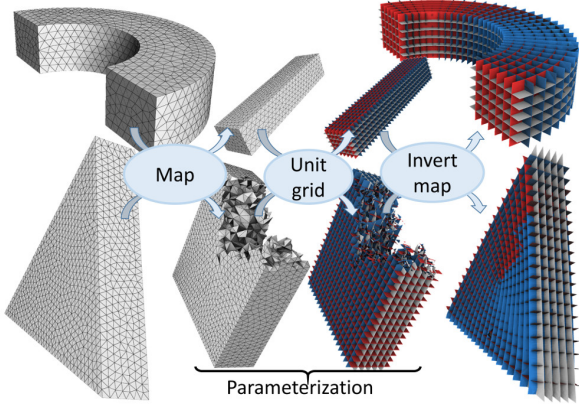


Figure 5: **Upper row:** A map deforms an input object (left) to align its normals with the (x, y, z) axis (middle-left). A regular grid (middle-right) defined on the image of the object can be deformed to match the original object using the invert function. **Lower row:** a global parameterization accepts a larger, more relaxed class of maps (grid preserving), that may include some discontinuities.

boundary is compatible with the quad facets of the hexahedra, the boolean difference is no longer a geometric operation, it becomes purely combinatorial.

- remesh the void (§§5.6).

2. Global parameterization

A global parameterization [9, 17, 12, 21, 7, 8] is a map that deforms a solid object in 3D space. Consider the unit grid in parameter space and project it onto the object space using the inverse of the deformation map, we can get a hexahedral mesh of the object (see Figure 5, upper row). In simple cases, the parameterization is a continuous map (e.g. in [9]), but it can also be discontinuous, as was shown in [17, 12]. If this discontinuous map satisfies “grid compatibility” constraints, then it can still produce a nice hexahedral mesh (see Figure 5, bottom row).

To generate a global parameterization, the main difficulty is to determine where to place the singularities i.e. the points of the volume where the pre-image of the unit grid is not a deformed grid. A common strategy is to compute a frame field that places some of these singularities, then to optimize a grid-compatible map guided by the frame field. In §8.2, we compare different solutions for both steps.

The output of the global parameterization is a set of local linear maps attached to (a subset of) the tetrahedra. The robustness of our pipeline is independent of the global parameterization quality. However, in order to produce hexahedra, whose support spreads over multiple tetrahedra, the parameterization has to be grid-compatible between pairs of adjacent tets.

3. Quad dominant mesh of the boundary

The objective in this section is to transform the original, triangulated, object boundary into a quad-dominant mesh that is compatible with the hexahedral mesh (see Figure 3).

The input of our algorithm is a tetrahedral mesh T with: a 3D global parameterization given by a (u, v, w) triplet attached to each tet corner, and a boolean flag per tet facet that indicates whether the facet has a valid parameterization.

The output is a quad-dominant remeshing of the boundary ∂T , that is a closed manifold surface free of self-intersection, as defined in §7. Moreover, we can ensure that the Hausdorff distance between ∂T and the quad dominant mesh is smaller than a given threshold.

Algorithm:. The input boundary ∂T is originally free of self-intersections. Our algorithm starts with this surface and transforms it into a compatible quad-dominant mesh by applying a series of local operations. The surface is guaranteed to remain free of self-intersections thanks to a rollback procedure that we apply after each step of the algorithm that modifies the surface geometry. This rollback procedure is described in the robustness discussion.

The whole quad-dominant remeshing algorithm is illustrated in Figure 6 and detailed in the rest of this section:

Figure 6.1: Extract the input surface ∂T and its 2D parameterization (§3.1);

Figure 6.2: create extra edges that are likely to match hexahedra edges (§3.2);

Figure 6.3: create charts of facets that correspond to the facets we want to create in the quad-dominant mesh, i.e. a set of quads that are likely to match a hexahedron facet (§3.3), or the original triangles elsewhere;

Figure 6.4: simplify the mesh by tentatively merging the facets that belong to the same chart. The (optional) control of the Hausdorff distance is performed during this step.

3.1. Extract the 2D parameterization of the boundary from the 3D parameterization

The 2D parameterization is defined as a restriction of the 3D parameterization to a subset of the boundary triangles. If exactly one coordinate has a constant integer value on a triangle, then the parameterization is defined on this triangle by the two other coordinates, else the triangle is flagged as invalid.

Figure 6-input shows three input scalar fields. The facets with an invalid 3D parameterization are shown in blue. The 2D parameterization Figure 6-1 has a valid 2D parameterization whenever it is drawn in red exactly once in the “Input” figure. The group of two triangles in blue

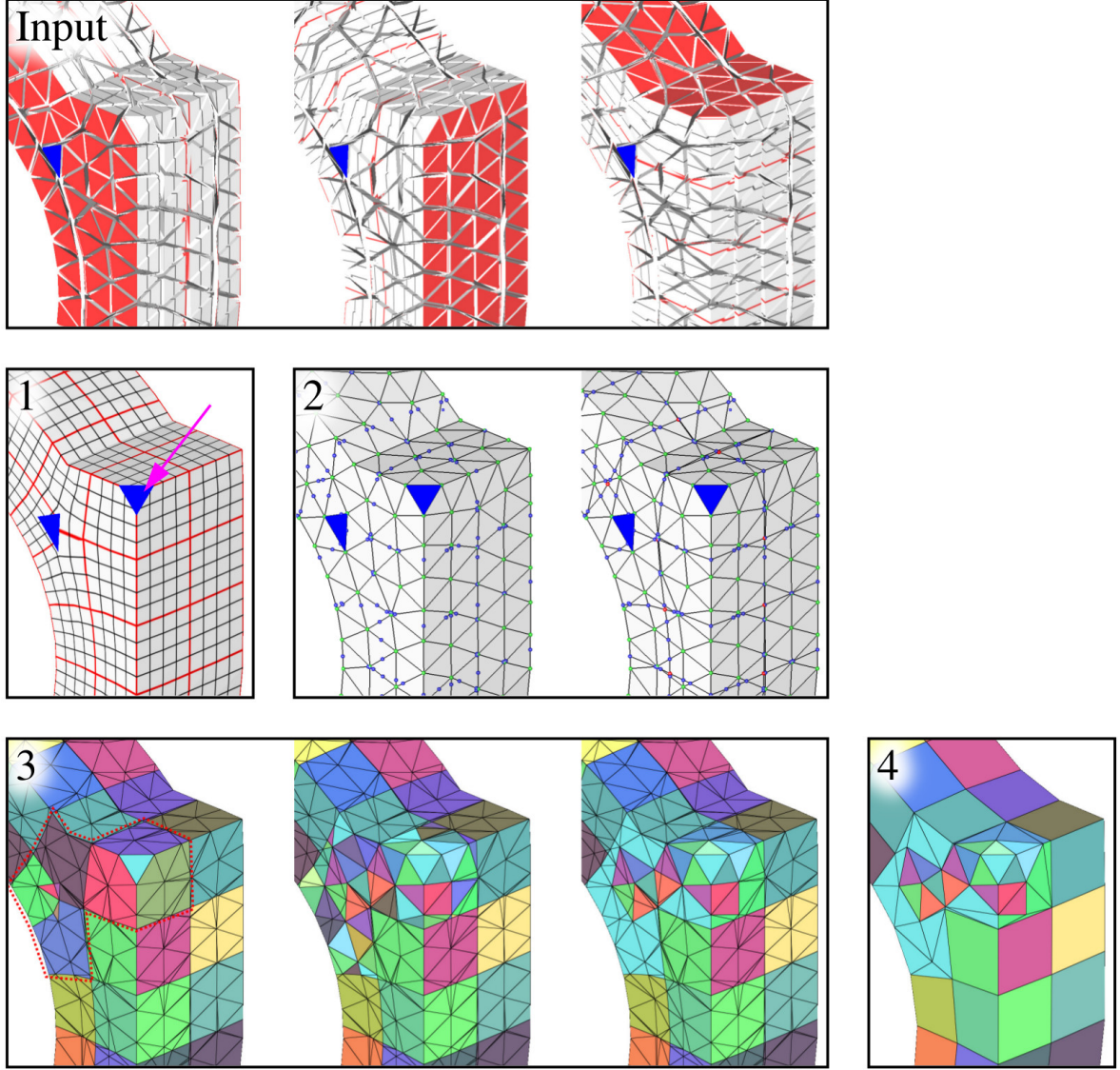


Figure 6: Compute a quad-dominant mesh of the input boundary. **(1)** Restrict the 3D parameterization to 2D; **(2)** imprint the integer isovalues of the parameterization into the mesh; **(3)** (left) regroup facets into charts separated by iso-values ; (middle) restore charts corresponding to the original triangulation in non-quad zone (red dotted area) ; (right) remove unnecessary constraints from the quad-triangles interface. **(4)** simplify the mesh w.r.t to the charts, producing the output quad-dominant mesh.

is invalid because one was already marked as invalid in the 3D parameterization, and the triangle in the corner (figure 6-1, marked by an arrow) is invalid because it has no constant iso-integer coordinate (never marked in red in the input).

3.2. Create new edges

This step imprints the integer isovalues of the 2D parameterization by splitting the facets. It is decomposed into two substeps (split the edges, split the facets), as illustrated in figure 6-2. This step is protected by a rollback procedure for each substep because the geometry of the object is modified by the numerical imprecision of vertices coordinates.

Split the edges. The first substep splits edges by introducing new vertices at integer values of the 2D parameterization coordinates. The left image of Figure 6-2 shows the resulting polygonal mesh. The blue vertices were added by splitting the edges. The original vertices are marked in green.

The algorithm that generates these new vertices works as follows: for each edge, we choose one of its neighboring polygons with a valid parameterization (if it exists) and use its parametric coordinates to split the edge as follows:

- we compute the barycentric coordinates (w.r.t the edge) of all points of the edge that have at least one integer coordinate in the parametric domain,
- we sort these points by the barycentric coordinates,

- we interpolate the geometric and parameterization coordinates on both adjacent polygons of the new points. The (integer) coordinate used that triggered the vertex is obviously not interpolated. We also snap other parametric coordinates that are closer than 10^{-10} to their nearest integer to account for numerical imprecisions in the grid preserving transition functions,
- we split the edge by introducing new vertices one by one. To avoid degeneracies, we only split the edge when: for each coordinate (geometric or parametric), the value of the new vertex is either strictly in the range of the values of the edge extremities, or equal to both values of the edge extremities.

At the end of the process, all coordinates (geometric or parametric) are either constant or monotonic along original edges, despite working with numerical imprecisions and snapping parametric coordinates.

Split the facets. We iteratively split facets by creating a new edge between a pair of vertices having a common integer parametric coordinate. The new edge is split by inserting new vertices at integer values of the other coordinates. This process creates edges along each integer isovalue, and introduces vertices at the intersections of integer isovalues, as illustrated in Figure 6-2 (right image, red vertices). Once all facets are split, we triangulate the facets.

Before splitting a facet, we check that:

- the new edge does not connect two successive vertices of the polygon (would produce a facet with 2 vertices),
- and, in parametric space, all the vertices of the new facets are located in the opposite half planes separated by the cut (except the extremities of the cut).

These conditions are sufficient to ensure that the algorithm terminates.

3.3. “Paint” the desired new mesh

Recall that the objective of this section is to produce an intersection-free mesh that has: quad facets where the parameterization is valid, the original triangles everywhere else, and a valid mesh at the junction of these regions.

Thanks to the imprint realized by the previous step, both the quads and the original triangles can be represented as groups of facets (that we call “charts”). The present step determines these charts as follows (Figure 6-3):

- **find quads:** We create one chart per facet then, for each pair of adjacent facets, we merge their charts if both have a valid parameterization and are not separated by an isovalue edge. We consider that a chart corresponds to a quad if its boundary solely contains isovalue edges and if it has 4 iso-value corners. Quads that are not flat or too distorted can be filtered, but it is optional (see Figure 10).

- **find triangles:** We create a chart per quad detected earlier, and a chart per facet not included in one of these quads. For each pair of adjacent facets, we merge their charts if they were produced from the same triangle of the original surface ∂T ;
- **relax the triangle constraint:** For each pair of adjacent facets, we merge their charts if they are not separated by an isovalue edge, and have one extremity located on a quad chart boundary. This step allows the mesh simplification algorithm to modify the original geometry nearby the quads.

3.4. Simplify the mesh

The objective here is to replace as many charts as possible by their corresponding facet. To do so, we visit each vertex that is not a quad corner, and try to remove it according to the number of charts it touches:

- 1 chart: we remove the incident fan of triangles and triangulate the hole (our implementation uses a min-weight triangulation);
- 2 charts: we remove the incident fan of triangles and triangulate the hole with the constraint that one edge should follow the chart boundary;
- more than 2 groups: we do not remove the vertex.

At the end of the process, most quad charts are represented by two triangles, and most original triangles that do not participate to quad charts are represented by a single triangle.

Controlling the Hausdorff distance. We ensure (Figure 7) that the symmetric Hausdorff distance between the boundary of the generated mesh and the boundary of the original mesh ∂T is lower than a user-defined ε by locking additional vertices before the rollback. The locked vertices are either the vertices that produced a facet of the new mesh with a point that is too far away from ∂T , or the three vertices of the triangle of ∂T with a point that is too far away from the new mesh. To evaluate an upper bound of the Hausdorff distance, we sample each facet in such a way that each point of the facet is closer than $\varepsilon/2$ to a sample, then we compute the distance between each sample and the other mesh. If this distance is greater than $\varepsilon/2$, the Hausdorff distance may be greater than ε , so the facet is considered as being too far away.

4. Hexahedral mesh

We would like the hexahedral mesh to be as large as possible, but under the constraint that it does not produce a void that would be impossible to remesh when combining it with the quad dominant mesh. We operate as follows:

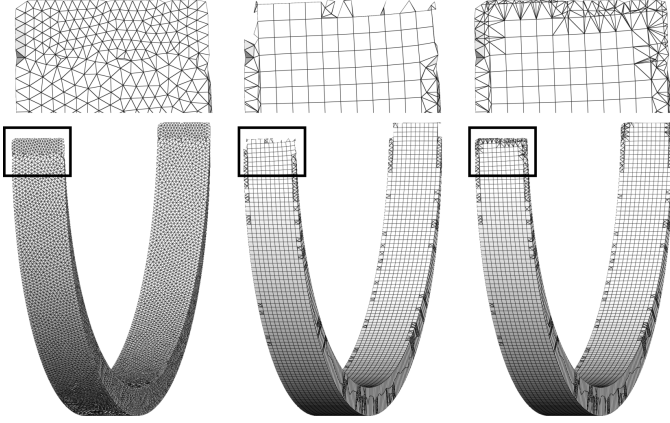


Figure 7: The quad dominant mesh computed from the input mesh (left) without (middle) and with (right) control of the symmetric Hausdorff distance.

- we construct a set of candidate hexahedra by extracting all possible hexahedra from the parameterization as done in [17],
- we merge position of vertices that are closer than the numerical precision error (10^{-10}). It allows to merge combinatorial elements instead of removing hexahedra due to geometric self-intersections. As the void boundary will be composed by both hexahedra faces and the quad-dominant mesh, all vertices to be merge must match geometrically. However, we only move hexahedra vertices because moving vertices of the quad-dominant mesh may produce unchecked self-intersections.
- we filter all hexahedra that would possibly produce self-intersections in the final void boundary. For each hexahedron that has an intersection with the quad-dominant mesh or another hexahedron: if this intersection is a vertex, an edge or a quad then the hexahedron is kept, otherwise, it is removed from the set of hexahedra. (Figures 9 and 8). Optionally, geometric criteria can be used to filter bad shape hexahedra (Figure 8-right).

As a result, the quad dominant mesh and the boundary of the hexahedral mesh exactly intersect on a set of facets, edges and vertices. In the next step, the boundary of the void will merge these primitives and have no self-intersections.

5. Void boundary

We define the boundary of the void volume as the difference between the quad-dominant mesh and the boundary of the produced hexahedra. At this stage, the quad-dominant mesh of the boundary and the extracted set of hexahedra are both free of intersection and compatible (the quad faces and hexahedra faces match), therefore the

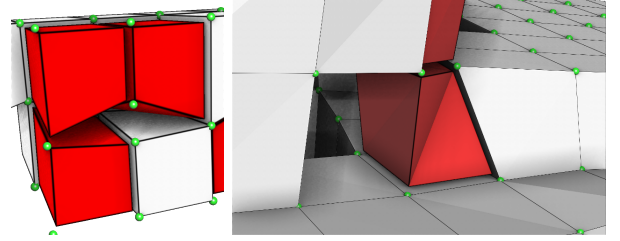


Figure 8: Examples of invalid configurations: hexahedra sharing 3 vertices (left) and a bad quality hexahedron (right).

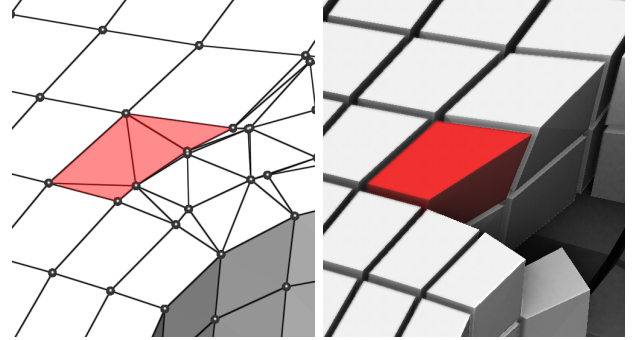


Figure 9: The quad-dominant mesh (on the left) does not match the facets of the red hexahedron, therefore it needs to be removed.

boolean operation that is required to compute the boundary of the void boils down to a purely combinatorial operation, and this step is straightforward to implement (in a certain sense, all difficulties have been pushed towards the previous steps).

6. Hexahedral dominant mesh

The last step of our pipeline fills the remaining void. The most robust solution is to build a constrained Delaunay tetrahedralization. We use TetGen [20] to do so.

It is also possible to plug algorithms that take into account the specific structure of the remaining void; *paving* [23] and *whisker-weaving* [22] present a (non-robust) option to explore.

7. Discussion on Robustness

We demonstrate the practical robustness of our method by running it without any failure on 200+ models, including challenging parameterisations as in Figure 10. The main difficulty is to ensure that the surface used to represent the boundary of the void can always be filled by tetrahedra.

7.1. Limitations of robustness

In practice, we only ensure that this surface is free of self-intersections.

We consider that a triangulated surface is self-intersecting if there exists a triangle that intersects another triangle in another way than sharing a vertex, an edge or

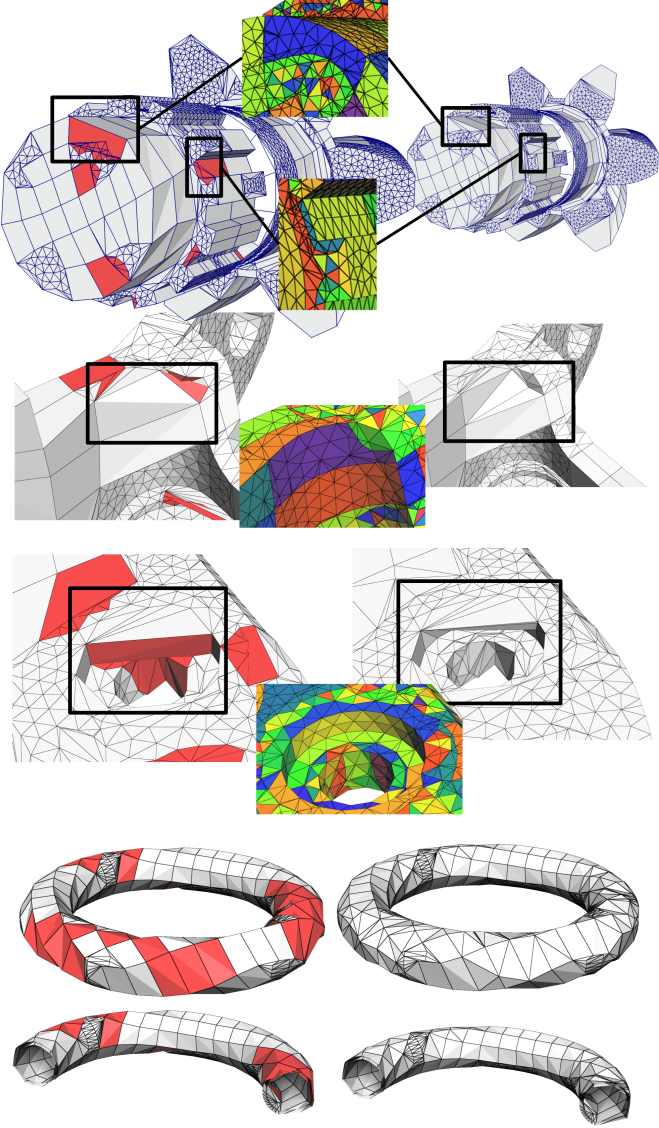


Figure 10: Extraction of quad-dominant boundary without our rollback procedure (left) produces self-intersections (red). Close-ups (middle) show the corresponding charts, and our result are on the right. A (non trivial) local remeshing strategy could fix some cases (upper left), but not intersections between faces that do not belong to the same part of the surface. The last model is an empty torus (clipped rendering) with many intersections between the remesh of its interior surface and its exterior surface.

the triangle with opposite orientation. This intersection is evaluated with exact predicates. When our surface contains quad facets, both possible triangulations of the quad are tested.

This is not sufficient to ensure that the void can be filled while working with a fixed precision floating point. While it is always possible to tetrahedralize a polyhedron if Steiner points are allowed, in practice it is possible to produce a Schönhardt polyhedron where there is not enough numerical precision to represent the position of the re-

quired Steiner point. Therefore, it is possible to produce a closed triangulated surface without self-intersections that can not be filled by tetrahedra with TetGen. This configuration is very unlikely: in tens of thousands of runs with different settings and models, we have never observed a TetGen failure when the input contained no self-intersection.

Note that the only assumption we make on the parameterization is that uvw -coordinates (when available) are valid floating-point numbers. As a consequence, it is possible that the imprint does not fit into memory because a triangle is too large in parametric space (for example, due to a bug).

7.2. Preventing self-intersections from appearing on the boundary of the void

One part of the boundary of the void is defined by remeshing the input surface, and the rest by the hexahedral mesh. We detail here how to avoid producing self-intersections in both parts.

7.2.1. Quad dominant remeshing of the object’s boundary

The boundary of the void is initialized as the boundary of the input tetrahedral mesh, thus initially it is free of no self-intersections. This property is preserved throughout all the mesh editing by the rollback strategy described below.

Rollback. We edit the mesh at different steps, and each step guarantees to produce a mesh without self-intersections thanks to a rollback procedure. All editing is made with atomic operations (edge split, facet split, vertex removal and triangle merge). During the mesh edition, we assign to each new facet the id of the primitive (edge, vertex and facet) that “triggered” the operation that produced the facet. At the end of the step, every facet that participates to an intersection locks the primitive that produced it. Then we re-run the process, without triggering operation on locked primitives, until no more intersections are found. The process is guaranteed to finish because the number of operations decreases at each iteration, and the in worst case we just keep the input mesh.

We use this rollback strategy at the end of each step:

- imprint 1 — edges splits: lock original edges that intersect, then rollback. Intersections are evaluated on a triangulation of the facets.
- imprint 2 — facets splits: lock original facets that have been subdivided into facets that participate to an intersection, then rollback.
- simplifying the mesh: lock the original vertices that have produced a facet with an intersection, then rollback.
- merge quads: lock the triangles that were merged into a quad that participates to an intersection, and rollback.

All these rollback operations guarantee that the quad dominant remeshing of the boundary of the surface has no self-intersections.

7.2.2. Boolean difference with the hexahedral mesh

The final boundary of the void is defined as the boolean difference between the quad dominant remeshing of the original surface and the boundary of the hexahedral mesh. The construction of the hexahedral mesh ensures that this final boundary of the void will be free of self-intersection. To do so, we start from all hexahedra extracted from the parameterization, and remove those that participate to an intersection (with the quad dominant mesh or other hexahedra). It ensures that the result will be free of self-intersections. To produce more hexahedra, we keep those who intersects exactly on a primitive (vertex, edge or quad), as we know that quads will cancel each other, and other primitives are supported by TetGen.

Note that prior to filtering, we adjust hexahedra vertices positions to make it easier to find exact matches between primitives, however we do not move vertices of the quad-dominant mesh, so this does not spoil the robustness.

8. Results and analysis

In this section, we compare our results to previous work (§8.1), we demonstrate that our pipeline makes it possible to compare different implementations of each step (§8.2), and we propose a straightforward adaptation to produce conformal meshes by introducing pyramids as interface elements between hexahedra and tetrahedra (§8.3). We experimented with our algorithm on a database of 200+ models, the reader can refer to the supplemental material for the complete list, images and hex-proportion statistics. General trends are summarized and analyzed below.

8.1. Positioning w.r.t the previous work

In this article we do not try to improve quality of individual elements, but rather we focus on the mesh extraction. Therefore, the quality of the elements we obtain is identical to the state of the art.

There are few articles with focus on the robust mesh extraction: polyhedral agglomeration, PGP3D and HexEx [21, 8, 13]. Let us recapitulate the main reasons why we propose a new method of mesh extraction:

- *PGP3D*: we obtain a more faithful mesh boundary. It is either the original mesh or the deformed grid, whereas [21] re-samples the surface and introduces new points until a Hausdorff distance criterion is met. Figure 11 shows that the results are sensitive to the distance threshold.
- *PGP3D*: hexahedra are directly extracted from the parameterization rather than recombined from a tetrahedral mesh generated from a point cloud. It prevents extracting isolated hexahedra and mismatches

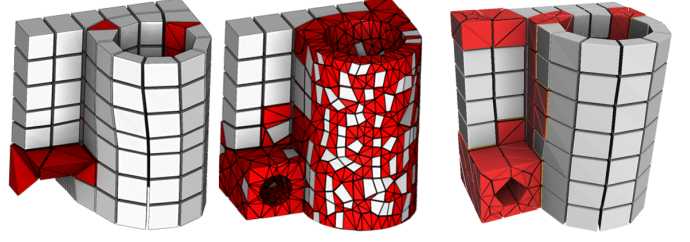


Figure 11: The mesh boundary produced by PGP3D [21] depends on a parameter that may miss some parts (left) or produce useless tetrahedra (middle). Our method is not subject to these problems (right).

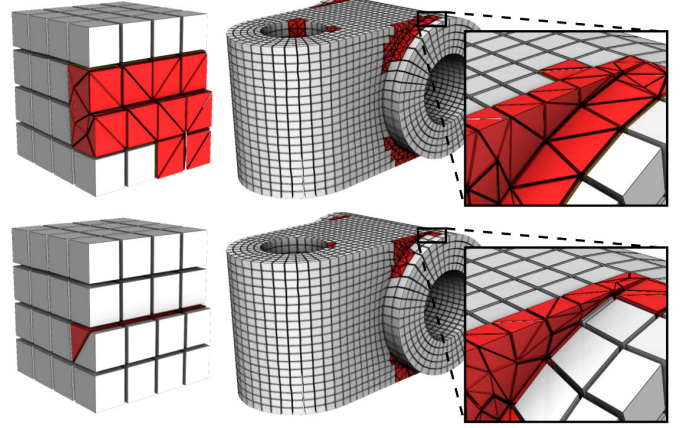


Figure 12: We preserve geometric details with tetrahedra. Cubecover and its extensions cannot directly handle them. **Top row:** direct tetrahedralization of the remaining void, **bottom row:** paving + whisker weaving used to fill the void.

that occur in [21, Figure 20] when the parameterization is distorted too much.

- *HexEx*: works only when a grid preserving parameterization is defined everywhere. It does not help facing problems of compatibility between hexahedral and tetrahedral zones in hex-dominant meshing.
- *HexEx* does not control the Hausdorff distance to the input mesh (Figures 1 and 12).
- *Polyhedral agglomeration*: this method may produce inverted elements, and there is no guarantee that all cases can be treated.
- *Polyhedral agglomeration*: suffers from the same problem than PGP: the parameterization is given by samples (not linear map), that may not be able to handle strong anisotropy or shear. Another drawback of the polyhedral agglomeration is the input mesh density, it cannot create a hex-dominant mesh with finer elements than the input mesh.

8.2. The pipeline as a benchmarking tool

As we mentioned before, our contribution is two-fold: first of all, it serves for a *robust mesh extraction*, and, due to its modular structure, the pipeline provides a *testing*

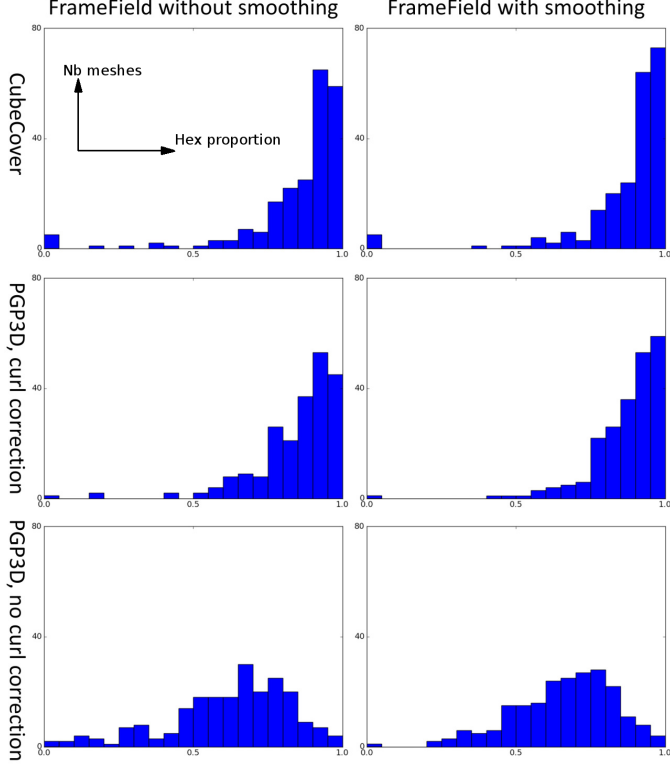


Figure 13: Distribution of the proportion of volume filled by hexahedra for all the 200+ models of the database (horizontal axis: proportion of hexahedra, vertical axis: number of models). In the first column, the frame field is obtained by solving a linear system. In the second column it is further optimized with non-linear iterations. The rows correspond to the global parameterization algorithm: CubeCover, PGP3D with curl correction, and PGP3D without curl correction respectively.

environment to benchmark each step separately. This section shows an example of such a benchmark.

Test frame fields. Let us say we want to evaluate impact of the frame field smoothing algorithm. [19] proposes two options: the first one requires only solving a least-squares problem, and the second one further improve this initialization with a non-linear optimization. The authors claim that the initializer only can suffice. The impact of both solutions is presented in the columns of Figure 13. We observe that it does not really impact the proportion of hexahedra except for a few models that really need the non-linear optimization. It confirms the observation done in [19] that the non-linear optimization is mandatory for objects of revolution (cylinders), but it is less important for other models (Figure 14).

Test the parameterization. Here we test three different global parameterization algorithms: CubeCover [17]² and

²Since we are aiming hex-dominant meshing, our default implementation runs [17] on the mesh **minus the frame field singularities** (defined as in [19]). Not including frame field singularities makes it easier to implement, and offers more degrees of freedom around singularities that are not compatible with hexahedral remeshing.

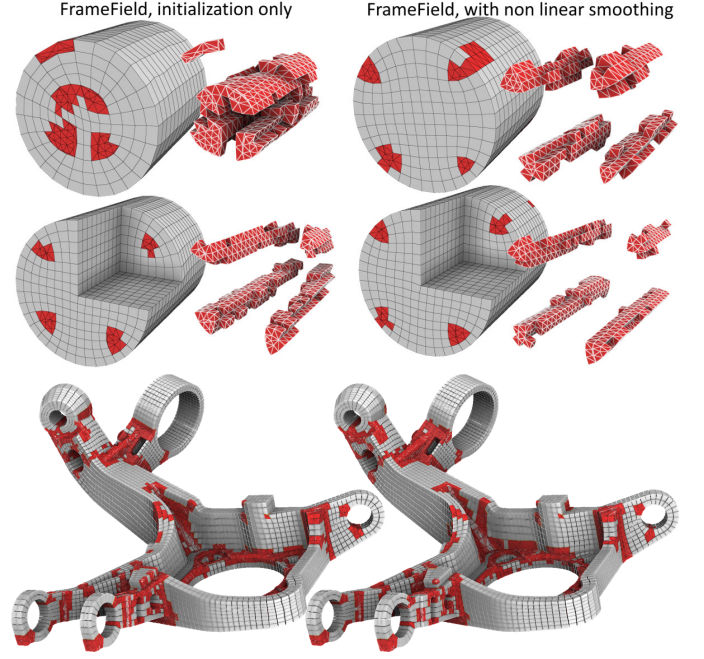


Figure 14: Impact of frame fields computed without (left column) or with (right column) the non linear part of the optimization. For the cylinders (top row), the remaining tetrahedra are visualized alone, to better reveal the inner structure.

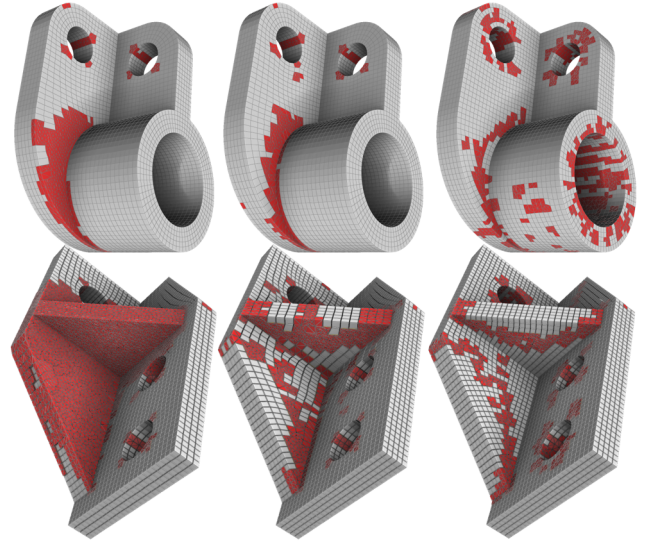


Figure 15: Two shapes remeshed by CubeCover, PGP3D with curl correction and PGP3D without curl correction.

PGP3D with and without curl correction [21].

The performance of each algorithm is shown in the rows of Figure 13. We observe that CubeCover and PGP3D with curl correction generate the same average proportion of hexahedra 87%, but with a better median for cube cover (93% versus 91%). PGP3D without curl correction obtains only 68% on average. The distribution of hexahedra on the model is very different (Figure 15): CubeCover may produce large remaining voids whereas PGP3D evenly distributes them over the mesh.

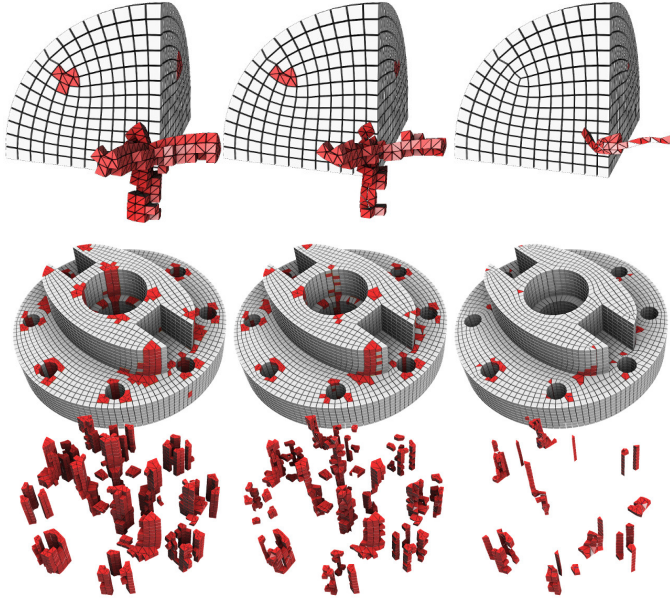


Figure 16: Hexahedral dominant meshes obtained by remeshing the void with (from left to right) TetGen, our adaptation of [4], and our paving + whisker weaving implementation. To better visualize the mesh inside the volume we also show the tetrahedra separately.

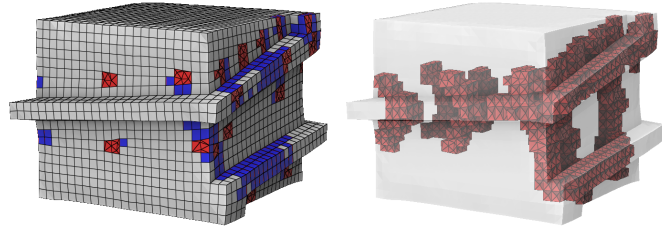


Figure 17: Conformal mesh with pyramids. Hexahedra are shown in gray, pyramids in blue and tetrahedra in red. **Left:** a pyramid is generated for each quad of the boundary of the void. **Right:** the remaining void is still filled by tetrahedra.

Test the final fill. The final stage of our pipeline is the meshing of the remaining void. The robust solution is given by TetGen, however with additional efforts the proportion of hexahedra can be significantly increased. Figure 16 shows an example of our adaptation of [4] and paving + whisker weaving. In general, when successful, paving + whisker-weaving performs better than [4], but is less robust (we observed many failure cases).

8.3. Conformal mesh with pyramids

Our pipeline can be adapted to generate conformal meshes by introducing a pyramid for each quad of the boundary of the void (Figure 17). In our pipeline, there is a simple solution to ensure that these pyramids will not produce self-intersections of the boundary of the void. Everytime we evaluate the intersection between a quad and another element (triangle, quad, or hexahedron), we also test whether other elements intersect the pyramid that would be generated by the quad.

9. Conclusion

In this article we presented a robust method of generating hex-dominant meshes free of degenerate or flipped elements. Our pipeline architecture allows to focus on each step independently, and to evaluate the impact of each algorithm. It also gives additional possibility of plugging-in specialized algorithms for meshing particular geometries, such as fillets or tubular structures.

Note that evaluation is another topic that will require further work. In the context of this article, we used for the evaluation the proportion of generated hexahedra, constrained to have a metric tensor condition number smaller than 10 (see §3.1). While it gives reasonably shaped elements, we think that a more thorough evaluation will be needed in the future, where the meshes are evaluated in the context of FEM (Finite Element Modeling) simulations. It would be important to compare the impact of the mesh on both the accuracy of the solution and speed of the simulation. Setting-up such an experiment requires a substantial amount of work, well beyond the scope of this article. It may be considered as a first step towards this direction, by providing an automatic and reproducible pipeline for the mesh generation part.

References

References

- [1] Steven E. Benzley, Karl Ernest Perry, Brett Clark Merkley, and Greg Sjaardema. A comparison of all-hexahedral and all-tetrahedral finite element meshes for elastic and elasto-plastic analysis. In *International Meshing Roundtable conf. proc.*, 1995.
- [2] Ted D. Blacker and Ray J. Meyers. Seams and wedges in plastering: A 3-d hexahedral mesh generation algorithm. *Engineering with Computers*, 9(2):83–93, 1993.
- [3] Nestor A Calvo and Sergio R Idelsohn. All-hexahedral element meshing: Generation of the dual mesh by recurrent subdivision. *Computer Methods in Applied Mechanics and Engineering*, 182(3–4):371 – 378, 2000.
- [4] Tristan Carrier-Baudouin, Jean-François Remacle, Emilie Marchandise, François Henrotte, and Christophe Geuzaine. A frontal approach to hex-dominant mesh generation. *Advanced Modeling and Simulation in Engineering Sciences*, 1(1), 2014.
- [5] R Choudhria and Philippe Véron. Identifying and re-meshing contact interfaces in a polyhedral assembly for digital mock-up. 22:47–58, 08 2006.
- [6] Brett Clark, Byron W. Hanks, and Corey Ernst. Conformal assembly meshing with tolerant imprinting. 01 2008.
- [7] Xianzhong Fang, Weiwei Xu, Hujun Bao, and Jin Huang. All-hex meshing using closed-form induced polycube. *ACM Trans. Graph.*, 35(4):124:1–124:9, July 2016.
- [8] X. Gao, W. Jakob, M. Tarini, and D. Panozzo. Robust hex-dominant mesh generation using field-guided polyhedral agglomeration. *ACM TOG (SIGGRAPH conf. proc.)*, 2017.
- [9] J. Gregson, A. Sheffer, and E. Zhang. All-hex mesh generation via volumetric polycube deformation. *Computer Graphics Forum (Special Issue of Symposium on Geometry Processing 2011)*, 30(5):to appear, 2011.
- [10] Jin Huang, Yiying Tong, Hongyu Wei, and Hujun Bao. Boundary aligned smooth 3d cross-frame field. *ACM Trans. Graph.*, 30(6):143:1–143:8, December 2011.
- [11] Tengfei Jiang, Jin Huang, Yuanzhen Wang, Yiying Tong, and Hujun Bao. Frame field singularity correction for automatic

- hexahedralization. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1189–1199, 2014.
- [12] Yufei Li, Yang Liu, Weiwei Xu, Wenping Wang, and Baining Guo. All-hex meshing using singularity-restricted field. *ACM Trans. Graph.*, 31(6):177:1–177:11, November 2012.
 - [13] Max Lyon, David Bommes, and Leif Kobbelt. HexEx: Robust hexahedral mesh extraction. *ACM Trans. Graph.*, 2016.
 - [14] Loïc Maréchal. A new approach to octree-based hexahedral meshing. In *International Meshing Roundtable conf. proc.*, 2001.
 - [15] Ray J. Meyers, Timothy J. Tautges, Philip M. Tuchinsky, and Dr. Philip M. Tuchinsky. The "hex-tet" hex-dominant meshing algorithm as implemented in cubit. In *in CUBIT; Proceedings, 7th International Meshing Roundtable 98*, pages 151–158. Sandia National Laboratories, 1998.
 - [16] M. Müller-Hannemann. Hexahedral mesh generation by successive dual cycle elimination. *Engineering with Computers*, 15(3):269–279, 1999.
 - [17] M. Nieser, U. Reitebuch, and K. Polthier. Cubecover– parameterization of 3d volumes. *Computer Graphics Forum*, 30(5):1397–1406, 2011.
 - [18] Steven J. Owen and Sunil Saigal. H-morph: an indirect approach to advancing front hex meshing. *International Journal for Numerical Methods in Engineering*, 49(1-2):289–312, 2000.
 - [19] Nicolas Ray, Dmitry Sokolov, and Bruno Lévy. Practical 3d frame field generation. *ACM Trans. Graph.*, 35(6):233:1–233:9, November 2016.
 - [20] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36, February 2015.
 - [21] Dmitry Sokolov, Nicolas Ray, Lionel Untereiner, and Bruno Lévy. Hexahedral-dominant meshing. *ACM Trans. Graph.*, 35(5):157:1–157:23, June 2016.
 - [22] T. J. Tautges, T. Blacker, and S. A. Mitchell. The whisker weaving algorithm: A connectivity-based method for constructing all-hexahedral finite element meshes. *International Journal for Numerical Methods in Engineering*, 39(19):3327–3349, 1996.
 - [23] David R. White and Paul Kinney. Redesign of the Paving Algorithm: Robustness Enhancements through Element by Element Meshing. In *Proceedings of the 6th International Meshing Roundtable*, pages 323–335, 1997.
 - [24] Yongjie Zhang and Chandrajit Bajaj. Adaptive and quality quadrilateral/hexahedral meshing from volumetric data. In *International Meshing Roundtable conf. proc.*, 2004.